Wiener Filtering Using Streaming SIMD Extensions

Version 2.1

01/99

Order Number: 243641-001

Table of Contents

Revision History

Revision	Revision History	Date
2.1	FCS revision.	01/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1. *The Pocket Handbook of Image Processing Algorithms in C*, by Harley R Myler and Aruthur R. Weeks. ISBN 0-13-642240-3.
2. *Increasing the Accuracy of the Results from the Reciprocal and Reciprocal Square Root Instructions using the Newton-Raphson Method*, Intel® Application Note (AP-803, Order Number: 243637-001).
3. *Split-Radix FFT*, Intel® Application Note (AP-808, Order Number: 243642-001).

1 Introduction

Streaming SIMD Extensions for the Intel® Architecture (IA) instruction set provide single precision floating point single-instruction, multiple-data (SIMD) instructions. These instructions provide a means to accelerate applications that rely heavily on floating-point operations, such as 3D geometry, video processing, some image processing, and spatial (3D) audio. This application note discusses Wiener Filtering, and includes examples of code that exploit the Streaming SIMD Extensions.

2 The Wiener Filter Algorithm

Wiener filtering (also known as Least Mean Square filtering) is a technique for removing unwanted noise from an image. The description of this algorithm is from *The Pocket Handbook of Image Processing Algorithms in C*, by Harley R Myler and Aruthur R. Weeks [1]. The algorithm has four (Fourier transformed) vector inputs, representing (one component of) the original image (Image), the degraded image (Guv), the noise image spectra (Noise) and the degradation function (Huv). Each input is a vector of row*col complex numbers. The complex numbers are represented as two contiguous floats for the real and imaginary parts of the number. An additional parameter, gamma, is included in the computation. When gamma is 1.0, the filter is known as non-parametric. The parameters to the filter can be adjusted until the filtered image is satisfactory.

2.1 Applications for Wiener Filters

Wiener filters are commomly used in image processing applications to remove noise from reconstructed images. Wiener filtering is often used to restore a blurry image. However, the Wiener filter has proved important in adaptive filtering, has been used for wavelet transforms, and has found application in communications and other DSP-related disciplines. The reader also should be aware that Fourier transformation is a key element in any signal processing discipline. Please refer to the Intel® Application Note, Split-Radix FFT (AP-808) for further information on implementing a Fourier transform using Streaming SIMD Extensions.

2.2 Implementing the Wiener Filter

As described in Section 2.1, the input to the function is four arrays of complex numbers. For each element of the image, the following operations are carried out using complex arithmetic. The complex variables D and Hs are intermediate variables used during the computation. The function `Complex_conj` is used to take the complex conjugate of a complex number. When divides occur, a check (using an if statement) must be done to ensure that the denominator is non-zero. In the event that a denominator is zero, the result should be set to zero.

1. Complex Noise = gamma * (Noise * Complex_conj (Noise))
2. Complex D = Image * Complex_conj (Image)
3. Complex D = Noise / D
4. Complex Hs = Huv * Complex_conj (Huv)
5. Complex Num = Complex_conj (Huv) * Guv

6. Complex Image = Num / (Hs + D)

3 Performance

First, the code can be optimized simply by observing that many of the operations involve multiplying a number by its complex conjugate. Since the result has no imaginary component, many of the operations specified in Section 2.2 can be simplified. The resultant C code is given in Section 5.

Before the code can be optimized for Streaming SIMD Extensions, it must be converted to a form suitable for SIMD execution. Four iterations of the original C code are gathered together and processed in a single iteration of the new loop. Each pass through the new loop does the work of four of the original iterations. This process, called "vectorization" allows the use of Streaming SIMD Extensions, and can provide a significant performance benefit.

The list of operations in Section 2.2 shows that three divides must be done per iteration: one divide in Step 3, (the imaginary component requires no divide since the imaginary part is zero), and two divides in Step 6. One of the latter two divides can be removed by noting that they both have the same denominator. The if statement required to check for a zero denominator can be removed by using a masking technique. Further improvements can be obtained by replacing all divides with reciprocal approximations. These techniques are described below.

After the code has been converted to a SIMD format, the checks for zero denominators (if statements) can be removed by creating a mask for the non-zero denominator elements, and ANDing that mask to the result of the division to zero out the elements where a division by zero occurred. This technique assumes that QNANs are being generated rather than SNANs, by masking SIMD floating-point exceptions in the MXCSR register, so that no floating point divide by zero exceptions will occur. For example, assume that you want to compute the quantity (N / D), where N and D are floats. A typical code sequence is given below.

```
If ( D != 0 )
    Result = N / D;
Else
    Result = 0.0;
```

The result can be computed as `(and ( div ( N, D ), cmp_neq ( D, 0 ) ) )`, a computation that does not require an if statement. Using intrinsics, this would be expressed as `mm_and_ps ( mm_div_ps ( N, D ), mm_cmpneq_ps ( D, zero ) )`.

This technique is specific to Streaming SIMD Extensions, and is implemented in the attached Intrinsics and assembly language versions of the code.

The Newton-Raphson method is a classic technique for approximating functions. The initial "guess" at the reciprocal is computed using the `rcpps` instruction. Subsequently, the "guess" is improved using the Newton-Raphson method. The result is not as accurate as what is provided by the divide instruction; however, it can be obtained significantly faster. (Programmers must determine if their application allows a reduced precision answer.) Full details and discussion of this technique are available in the Newton-Raphson application note [2]. The specific code sequence employed in this Wiener filter is given below. (The denominator should be examined to be certain a division by zero is not occurring.)

```
RC = _mm_rcpps( D );
RECIP = _mm_sub( _mm_add( RC, RC ), _mm_mul( RC, _mm_mul( RC, D ) ) );
```

4 Conclusion

Streaming SIMD Extensions can provide a significant performance improvement for the Wiener filter algorithm, compared to coding in C. The improvements cited in this document are the result of several techniques. The techniques include using Streaming SIMD Extensions (vectorizing the code), and removing conditional branch instructions (if statements) by using masking operations provided by Streaming SIMD Extensions. If it is acceptable to reduce the numerical precision of the result, then a further gain can be realized by replacing divide operations with reciprocal approximations (employing a Newton-Raphson technique).

5 C Coding Example

```
/*
 * Wiener Filter (also known as the Least Mean Square filter)
 *
 *  Reference: The Pocket Handbook of Image Processing Algorithms in C
 *             by Harley R Myler & Arthur R. Weeks
 *             1993 Prentice-Hall, ISBN 0-13-642240-3 p260-3.
 *
 * The data is several arrays of complex floats in row major order.
 * The description for the algorithm from p260 states:
 *
 *    The algorithm computes a parametric Wiener filter on the
 *    Fourier transform of a degraded image, Guv, with noise
 *    spectra N, degradation function Huv, and original image Img.
 *    The computation is in place, so that the filtered version of
 *    the input is returned in the original image variable.  The
 *    original and noise images are either estimations form some
 *    predictive function or ad hoc approximations.  If the noise
 *    image is zero, the process reduces to the inverse filter.
 *
 *    The Weiner parameter gamma is passed to the algorithm.
 *    If this parameter is 1.0, the filter is non-parametric.
 *    Methods exist in the literature to derive the parameter value;
 *    however, it is sometimes determined from trial and error.
 *
 *NOTE!!!! The code on page 263 has an error.  In cxml, the complex
 * multiply routine, the imaginary part of the computation should be
 * a*d + b*c, not a*d - b*c.
```

```
 *
 *NOTE! (another error) The *complex* array length is rows*cols, so the
 * *float* array length should be 2*rows*cols.  Also, note that the
 * algorithm operates on one component of the pixel.
 */
void    wiener_filter ( float *Img,
                        float *Huv,
                        float *No,
                        float *Guv,
                        float  gamma,
                        int    rows,
                        int    cols)
{
  int   i, sz;
  float numr, numi, dr, hsr;

  sz = 2 * rows * cols;
  for (i = 0; i < sz; i += 2)
  {
    /* Compute (in place) the noise spectral density with Wiener gamma*/

    No[i]   = (float) ( gamma * ( No[i]*No[i] + No[i+1]*No[i+1] ) );
    No[i+1] = (float) 0.0;

    /* Compute image spectral density */

    dr   = (float) ( Img[i]*Img[i] + Img[i+1]*Img[i+1] );

    /* Compute denominator spectral density term */

    if (dr != 0.0)
      dr = (float)  (No[i] / dr) ;


    /* Compute degradation power spectrum */

    hsr  = (float) ( Huv[i]*Huv[i] + Huv[i+1]*Huv[i+1] );

    /* Compute numerator term */

    numr = (float) ( Huv[i]*Guv[i]   + Huv[i+1]*Guv[i+1] );
    numi = (float) ( Huv[i]*Guv[i+1] - Huv[i+1]*Guv[i  ] );
```

```
    /* Final computation */

    if ( (hsr + dr) != 0.0 )
    {
      Img[i]   = (float) (numr / (hsr + dr));
      Img[i+1] = (float) (numi / (hsr + dr));
    }
    else
    {
      Img[i]   = (float) 0.0;
      Img[i+1] = (float) 0.0;
    }
  }
} /* wiener_filter */
```

6 Streaming SIMD Extensions Assembly Code Example

Sections 6.1 and 6.2 provide code samples for a Wiener filter, implemented using assembly instructions using Streaming SIMD Extensions and new intrinsics supported by the Intel® C/C++ Compiler. In each version, the Newton-Raphson reciprocal approximation replaces the divide instructions. The area where the divides were removed is indicated in the source code.

6.1 Assembly Code

```
#include <assert.h>
void    XMM_wiener_rcp( float *Img,
                        float *Huv,
                        float *No,
                        float *Guv,
                        float  gamma,
                        int    rows,
                        int    cols )
{
  assert(    (2 * rows * cols) > 3  );
  assert( !( (2 * rows * cols) & 3) );
  assert( !( ((int)Img) & 15 )      );         /* Assume alignment */
  assert( !( ((int)Huv) & 15 )      );
  assert( !( ((int)No)  & 15 )      );
  assert( !( ((int)Guv) & 15 )      );

  __asm
  {
```

```
  ;
  ; Register assignments:
  ;
  ;   ecx    loop count
  ;   esi    No
  ;   edx    Guv
  ;   eax    Huv
  ;   edi    Img
  ;
  ;   xmm7   gamma
  ;   xmm6   nr4, then dr4, then hsr4+dr4
  ;   xmm5   zero
  ;
  ;   xmm0-4 scratch
  ;
  ;
  ;for (i = 0; i < sz; i += 8)
  ;{
  ;

  xorps         xmm5, xmm5      ; create [ 0, 0, 0, 0 ]

  mov           ecx, rows
  imul          ecx, cols       ; compute array size (in complex num)
  sal           ecx, 3          ; 8 bytes per complex num
  mov           esi, No         ; base of No array (noise)
  mov           edx, Guv        ; base of Guv array (degraded image)
  mov           eax, Huv        ; base of Huv  array (degradation function)
  mov           edi, Img        ; base of Img array (image)

  add           esi, ecx        ; point to end of arrays...
  add           edx, ecx
  add           eax, ecx
  add           edi, ecx

  neg           ecx             ; array end - count = array base

  movss         xmm7, gamma
  shufps        xmm7, xmm7, 0   ; create [ gamma, gamma, gamma, gamma ]


forloop:
```

```
;/*
; * Compute (in place) the noise spectral density with Wiener gamma
; *
; * complex Noise = gamma * (Noise * complex conj Noise)
; *
; * No[i]   = (float) ( gamma * ( No[i]*No[i] + No[i+1]*No[i+1] ) );
; * No[i+1] = (float) 0.0;
; */

movaps      xmm0, [esi + ecx]
movaps      xmm1, [esi + ecx + 4*4]
movaps      xmm2, xmm0
shufps      xmm2, xmm1, 0x88        ; xmm2  = nor4
shufps      xmm0, xmm1, 0xdd        ; xmm0  = noi4

                                    ; hoisted code (see below)
movaps      xmm3, [edi + ecx]       ; get Img[i]
movaps      xmm4, [edi + ecx + 4*4] ; get Img[i+4]

mulps       xmm2, xmm2
mulps       xmm0, xmm0
addps       xmm0, xmm2

                                    ; hoisted code (see below)
movaps      xmm2, xmm3              ;
shufps      xmm2, xmm4, 0x88        ; xmm2 = inr4
shufps      xmm3, xmm4, 0xdd        ; xmm3 = ini4


mulps       xmm0, xmm7
movaps      xmm1, xmm0
movaps      xmm6, xmm0              ; xmm6, xmm1, xmm0 = nr4

unpcklps    xmm0, xmm5
unpckhps    xmm1, xmm5
movaps      [esi + ecx],       xmm0
movaps      [esi + ecx + 4*4], xmm1


;/*
; * Compute image spectral density
; *
; * Complex D = Image * complex conj Image
```

```
; *
; * dr = (float) ( Img[i]*Img[i] + Img[i+1]*Img[i+1] );
; */
;
; Hoist the Img loads to exploit idle pipe
; Hoist the shuffles along with


                                    ; hoisted code (see below)
movaps      xmm0, [eax + ecx]       ; get Huv[i]...
movaps      xmm1, [eax + ecx + 4*4] ; get Huv[i+4]...


mulps       xmm2, xmm2
mulps       xmm3, xmm3
addps       xmm3, xmm2              ; xmm3 = dr4

;
; start prefetch for next iteration...
;
prefetchnta [esi + ecx + 32]
prefetchnta [edi + ecx + 32]
prefetchnta [eax + ecx + 32]
prefetchnta [edx + ecx + 32]


;/*
; * Compute denominator spectral density term
; *
; * Complex D = noise / D
; *
; * if (dr != 0.0)
; *   dr = (float)  (No[i] / dr) ;
; */
; replace the divide with an approximation
;

                                    ; hoisted code (see below)
movaps      xmm2, xmm0              ; make copy of Huv[i].....
shufps      xmm2, xmm1, 0x88        ; xmm2 = hr4
shufps      xmm0, xmm1, 0xdd        ; xmm0 = hi4

rcpps       xmm1, xmm3              ; start the approximation
```

```
cmpneqps    xmm5, xmm3              ; do the masks first
                                    ; (restore xmm5 later)

mulps       xmm3, xmm1              ; improve the approximation
mulps       xmm3, xmm1
addps       xmm1, xmm1
subps       xmm1, xmm3              ; approximation complete

                                    ; hoisted code (see below)
movaps      xmm4, xmm0              ; xmm4 = hi4
movaps      xmm3, xmm2              ; xmm3 = hr4
mulps       xmm2, xmm2
mulps       xmm0, xmm0
addps       xmm0, xmm2              ; xmm0 = hsr4


mulps       xmm6, xmm1              ; complete the division
                                    ; xmm6 = dr4
andps       xmm6, xmm5
xorps       xmm5, xmm5              ; restore the zero in xmm5


;REPLACED CODE-- reciprocal aproximation code is above
;divps      xmm6, xmm3
;cmpneqps   xmm3, xmm5
;andps      xmm6, xmm3              ; xmm6 = dr4



;/*
; * Compute degradation power spectrum
; *
; * Complex Hs = Huv * complex conj Huv
; *
; * hsr  = (float) ( Huv[i]*Huv[i] + Huv[i+1]*Huv[i+1] );
; */
; Hoist the loads of Huv to exploit an idle pipe
; Also, hoist the shuffles for same reason
; Hoist the movs as well.
;
addps       xmm6, xmm0              ; xmm6 = hsr4 + dr4
                                    ; (since we only need the
```

```
                                        ; sum of hsr4+dr4, compute it now)


;/*
; * Compute numerator term
; *
; * Complex Num = complex conj Huv * Guv
; *
; * numr   = (float) ( Huv[i]*Guv[i]   + Huv[i+1]*Guv[i+1] );
; * numi   = (float) ( Huv[i]*Guv[i+1] - Huv[i+1]*Guv[i  ] );
; */
movaps      xmm1, [edx + ecx]       ; get Guv[i]
movaps      xmm0, [edx + ecx + 4*4] ; get Guv[i+4]

movaps      xmm2, xmm1
shufps      xmm2, xmm0, 0x88        ; xmm2 = gr4
shufps      xmm1, xmm0, 0xdd        ; xmm1 = gi4


movaps      xmm0, xmm2
mulps       xmm2, xmm3              ; (hr4 * gr4)
mulps       xmm0, xmm4              ; (hi4 * gr4)

mulps       xmm3, xmm1              ; (hr4 * gi4)
mulps       xmm4, xmm1              ; (hi4 * gi4)

addps       xmm2, xmm4              ; xmm2 = numr4
subps       xmm3, xmm0              ; xmm3 = numi4

;/*
; * Final computation
; *
; * Complex Image = Num / (Hs + D)
; *
; * if ( (hsr + dr) != 0.0 )
; * {
; *   Img[i]   = (float) (numr / (hsr + dr));
; *   Img[i+1] = (float) (numi / (hsr + dr));
; * }
; * else
; * {
; *   Img[i]   = (float) 0.0;
; *   Img[i+1] = (float) 0.0;
```

```
    ; * }
    ; */
    ;replace these divides with approximations
    ;also, we get a bonus-- one reciprical and two multiplies
    ;in place of two divides.
    ;

    rcpps       xmm1, xmm6              ; start the approximation

    cmpneqps    xmm5, xmm6

    mulps       xmm6, xmm1              ; improve the approximation
    mulps       xmm6, xmm1
    addps       xmm1, xmm1
    subps       xmm1, xmm6              ; approximation complete

    mulps       xmm2, xmm1              ; complete the divisions
    mulps       xmm3, xmm1

    ;REPLACED CODE-- reciprocal aproximation code is above
    ;divps      xmm2, xmm6
    ;divps      xmm3, xmm6

    andps       xmm2, xmm5              ; xmm2 = inr4
    andps       xmm3, xmm5              ; xmm3 = ini4
    xorps       xmm5, xmm5              ; restore the zero in xmm5

    movaps      xmm0, xmm2

    unpcklps    xmm2, xmm3
    unpckhps    xmm0, xmm3

    movaps      [edi + ecx ],      xmm2
    movaps      [edi + ecx + 4*4], xmm0

    add         ecx, 32              ; 32bytes is 8 floats is 4 complex nums
    jne         forloop
  ;}

loopdone:

  }
} /* XMM_wiener_rcp */
```

6.2 Intrinsics Code

```
#include <iostream.h>
//#define MM_FUNCTIONALITY
#include <xmmintrin.h>
#include <assert.h>

void intrin_wiener_rcp( float *Img,
                        float *Huv,
                        float *No,
                        float *Guv,
                        float  gamma,
                        int    rows,
                        int    cols )
{
  int   i, sz;
  __m128 first2, next2, nor4, noi4, nr4, inr4, ini4, dr4;
  __m128 hr4, hi4, hsr4, gr4, gi4, numr4, numi4;
  __m128 rc,  denom;
  __m128 zero = _mm_set_ps1 (0.0);

  sz     = 2 * rows * cols;
  assert(  (sz > 3) & !(sz & 3)  );
  assert(  !( ((int)Img) & 15 )  );          /* Assume alignment */
  assert(  !( ((int)Huv) & 15 )  );
  assert(  !( ((int)No)  & 15 )  );
  assert(  !( ((int)Guv) & 15 )  );


  for (i = 0; i < sz; i += 8)
  {
    /*
     * Prefetch all the data for next loop iteration
     */
    _mm_prefetch( (char *) &No [i+8], _MM_HINT_NTA );
    _mm_prefetch( (char *) &Img[i+8], _MM_HINT_NTA );
    _mm_prefetch( (char *) &Huv[i+8], _MM_HINT_NTA );
    _mm_prefetch( (char *) &Guv[i+8], _MM_HINT_NTA );
    /*
     * Compute (in place) the noise spectral density with Wiener gamma
     *
     * complex Noise = gamma * (Noise * complex conj Noise)
     *
```

```
 * No[i]   = (float) ( gamma * ( No[i]*No[i] + No[i+1]*No[i+1] ) );
 * No[i+1] = (float) 0.0;
 */

first2  = _mm_load_ps   ( &No[i]   );
next2   = _mm_load_ps   ( &No[i+4] );
nor4    = _mm_shuffle_ps( first2, next2, 0x88 );
noi4    = _mm_shuffle_ps( first2, next2, 0xdd );

nr4     = _mm_mul_ps    ( _mm_set_ps1( gamma ) ,
                          _mm_add_ps ( _mm_mul_ps( nor4 , nor4 ),
                                       _mm_mul_ps( noi4 , noi4 )  )  );

_mm_store_ps( &No[i  ], _mm_unpacklo_ps ( nr4, zero ) );
_mm_store_ps( &No[i+4], _mm_unpackhi_ps ( nr4, zero ) );

/*
 * Compute image spectral density
 *
 * Complex D = Image * complex conj Image
 *
 * dr = (float) ( Img[i]*Img[i] + Img[i+1]*Img[i+1] );
 */
first2  = _mm_load_ps   ( &Img[i]   );
next2   = _mm_load_ps   ( &Img[i+4] );
inr4    = _mm_shuffle_ps( first2, next2, 0x88 );
ini4    = _mm_shuffle_ps( first2, next2, 0xdd );
dr4     = _mm_add_ps    ( _mm_mul_ps( inr4 , inr4),
                          _mm_mul_ps( ini4 , ini4)   );

/*
 * Compute denominator spectral density term
 *
 * Complex D = noise / D
 *
 * if (dr != 0.0)
 *   dr = (float)  (No[i] / dr) ;
 *
 * Do that reciprical division thing!
 */

rc = _mm_rcp_ps(dr4);
rc = _mm_sub_ps( _mm_add_ps( rc, rc),
```

```
                _mm_mul_ps( rc, _mm_mul_ps( rc, dr4) ) );

dr4 =  _mm_and_ps ( _mm_mul_ps   ( nr4 , rc  ),
                    _mm_cmpneq_ps( dr4, zero )   );


/*
 * Compute degradation power spectrum
 *
 * Complex Hs = Huv * complex conj Huv
 *
 * hsr  = (float) ( Huv[i]*Huv[i] + Huv[i+1]*Huv[i+1] );
 */
first2  = _mm_load_ps   ( &Huv[i]   );
next2   = _mm_load_ps   ( &Huv[i+4] );
hr4     = _mm_shuffle_ps( first2, next2, 0x88 );
hi4     = _mm_shuffle_ps( first2, next2, 0xdd );
hsr4    = _mm_add_ps    ( _mm_mul_ps (hr4 , hr4 ),
                          _mm_mul_ps (hi4 , hi4 )  );


/*
 * Compute numerator term
 *
 * Complex Num = complex conj Huv * Guv
 *
 * numr   = (float) ( Huv[i]*Guv[i]   + Huv[i+1]*Guv[i+1] );
 * numi   = (float) ( Huv[i]*Guv[i+1] - Huv[i+1]*Guv[i  ] );
 */
first2  = _mm_load_ps   ( &Guv[i]   );
next2   = _mm_load_ps   ( &Guv[i+4] );
gr4     = _mm_shuffle_ps( first2, next2, 0x88 );
gi4     = _mm_shuffle_ps( first2, next2, 0xdd );
numr4   = _mm_add_ps    ( _mm_mul_ps (hr4 , gr4),
                          _mm_mul_ps (hi4 , gi4)  );
numi4   = _mm_sub_ps    ( _mm_mul_ps (hr4 , gi4),
                          _mm_mul_ps (hi4 , gr4)  );
/*
 * Final computation
 *
 * Complex Image = Num / (Hs + D)
 *
 * if ( (hsr + dr) != 0.0 )
```

```
     * {
     *   Img[i]   = (float) (numr / (hsr + dr));
     *   Img[i+1] = (float) (numi / (hsr + dr));
     * }
     * else
     * {
     *   Img[i]   = (float) 0.0;
     *   Img[i+1] = (float) 0.0;
     * }
     *
     * Do the reciprical division thing
     */

    denom = _mm_add_ps( hsr4, dr4 );
    rc    = _mm_rcp_ps(denom);
    rc    = _mm_sub_ps( _mm_add_ps( rc, rc),
                        _mm_mul_ps( rc, _mm_mul_ps( rc, denom) ) );


    inr4  = _mm_and_ps( _mm_mul_ps   ( numr4 , rc ) ,
                        _mm_cmpneq_ps( denom, zero )   );

    ini4  = _mm_and_ps( _mm_mul_ps   ( numi4 , rc ) ,
                        _mm_cmpneq_ps( denom, zero )   );

    _mm_store_ps( &Img[i  ], _mm_unpacklo_ps ( inr4, ini4 ) );
    _mm_store_ps( &Img[i+4], _mm_unpackhi_ps ( inr4, ini4 ) );
  }
} /* intrin_wiener_rcp */
```